Summer 7-1-1975

A Data Base System Designed for Flexibility and Usability from FORTRAN ; CU-CS-072-75

Leon J. Osterweil
University of Colorado Boulder

Lori Clarke
University of Colorado Boulder

David W. Smith
University of Colorado Boulder

A Data Base System Designed for Flexibility and Usability from FORTRAN *

by

Leon Osterweil
Lori Clarke
David W. Smith

Department of Computer Science
University of Colorado
Boulder, Colorado 80302

#CU-CS-072-75 July 1975

CU-CS-072-75 is a condensed version of CU-CS-052-74

* This work supported by NSF Grant GJ-36461.

B

SUMMARY

This paper discusses the design, implementation and usefulness of a system for the flexible creation, accessing, and reformatting of list structured data bases in ANSI FORTRAN programs. The system is portrayed as being a useful tool in building prototype systems where frequent design modifications are expected and which, for reasons such as portability, are to be written in FORTRAN.

Key Words: Data Base, Data Management, Portability.

Introduction

The data base management system described here provides a flexible and powerful mechanism for creating and using list structured data bases from ANSI FORTRAN programs and for referencing items in such data bases symbolically. By a list structured data base, we mean a collection of lists of data aggregates (which we shall call nodes), where a data aggregate (node) consists of individual data items (fields) arranged in memory in some rigid predefined order and format. Many contemporary languages allow for the creation and symbolic referencing of items in such data aggregates. For example record formats, which can be declared in the DATA DIVISION of a COBOL program, can be used to implement such lists and nodes, and other languages such as PL/1 and PASCAL offer similar capabilities. The lengths of lists in such structures, however, must usually be prespecified, thereby limiting the flexibility of programs using them. ANSI FORTRAN unfortunately makes no explicit provision at all for either the efficient creation or graceful manipulation of lists of symbolically referenceable nodes. It is, however, frequently quite useful in tasks such as data management to be able to at least create such structures.

Our motivation for creating this system arose during a research effort involving the construction of a prototype programming system whose data structures and list organizations would ultimately be dictated at least in part by preliminary experience with the partially constructed prototype. Hence we realized that we needed a capability through which we could routinely design new data aggregates and routinely

alter and augment existing structures and list lengths without having to make modifications to an ever growing body of programs. We believe that the need for such a capability generally arises during most prototype development efforts, where the format of needed data structures cannot be precisely known at the outset. Because ANSI FORTRAN lacks this flexibility, we believe that FORTRAN's usefulness as a tool in software experimentation is weakened. Other considerations, such as portability, however, dictated the use of FORTRAN as our prototype development language. Thus we set about enabling the needed data aggregate creation flexibility while, nevertheless, leaving the existing language unaltered.

We restricted ourselves to creating and accessing symbolically referenceable nodes and fields and have not addressed ourselves to the problems of creating capabilities for performing the powerful operations on structures which exist in such languages as COBOL and PL/1. Such operations can easily be implemented as sets of higher level subroutines which call the lower level ones created and described here.

This system actually consists of 1) an initialization program which sets up the framework for building the data base and node structures according to specifications supplied by the user, 2) a library of user callable subroutines for doing such things as adding nodes to lists, and entering and retrieving data items into and from fields, and 3) a rollin/rollout package for saving data bases on mass storage files and then restoring them, perhaps after an extended period of time during which the node and field formats may have been changed.

The system allows the user to reference all nodes and fields by node and field names which he has selected himself, provided these

names are legal FORTRAN variable names.

The field names and node names which the user wishes to use from his program are first set up as an input file. The initialization program then processes this input file to create node formats. It builds tables which associate the user names with internal specifications of nodes and fields, and then writes these tables out to a binary data file on mass storage. It also creates labelled COMMON statements containing the field names, and node names. These COMMON statements are written to a different mass storage file consisting of symbolic line images. The COMMON statements are read from this symbolic file and incorporated into the user's programs during a preprocess phase of his execution. The tables are read from the binary file during an initiation phase of execution.

In order to change node descriptions, the user need only change his input file and rerun the initialization routine. New COMMON statements and new node formats, embodied in new access tables, will be created by the initialization routine and placed into the two mass storage files for use by all subsequent runs. Of course, all user subprograms which refer to nodes and fields must also be recompiled. Because all fields are referenced by the user merely by passing names to the data accessing routines, which in turn perform their accessing by using the tables created by the initialization program, old programs will continue to run successfully. Old data bases can be restructured using the rollout/rollin programs so that they also conform to the new data base description. Hence the crucial ideas here are to assure that all list and field references are made by means of variables, and that the values of the variables are reset only by automated

routines. This gives great flexibility, and makes for more natural, readable code since the node and field names can be made mnemonic.

Throughout this work we have attempted to make portability an important consideration. All routines are written in FORTRAN, and uses of nonstandard formations have been kept to a minimum and quarantined to a very small body of small subprograms. Word and field sizes are initialized once in a BLOCK DATA subprogram in an effort to aid machine independence.

A detailed description of the actual implementation of this system can be found in [2].

Overview

The data base system allows for the creation and manipulation of lists organized in two different ways -- sequentially and linked. A list is basically an ordered collection of nodes. The two organizations differ in the mechanisms by which the successor node of a given node can be reached (see [1], Section 2.2). Because each kind of list structure is well suited to a different class of important problems, it was deemed important to give users both kinds of list capabilities.

Users create lists by first defining node formats, and specifying whether a given node type is to be a part of either a linked list or a sequential list. System utility routines can then be used to compose nodes into lists. All sequential lists built within the system will be composed entirely of nodes of the same type (format). Moreover for each sequential node type there can be only one list composed of these nodes. On the other hand, it is possible to create many different linked lists, each composed of nodes of varying linked node types.

All lists comprising a list structured data base reside in a large FORTRAN vector called the data base area. The data base area is divided into three sections; the prefix, the sequential list area, and the linked list area (see Figure 1). The prefix occupies a small area at the beginning of the data base area. It contains information specific to the structure of the particular data base area containing it. For example, the size of the data base area, and the location and sizes of the various sequential lists are kept in the prefix. The prefix allows each list structured data base to be more or less

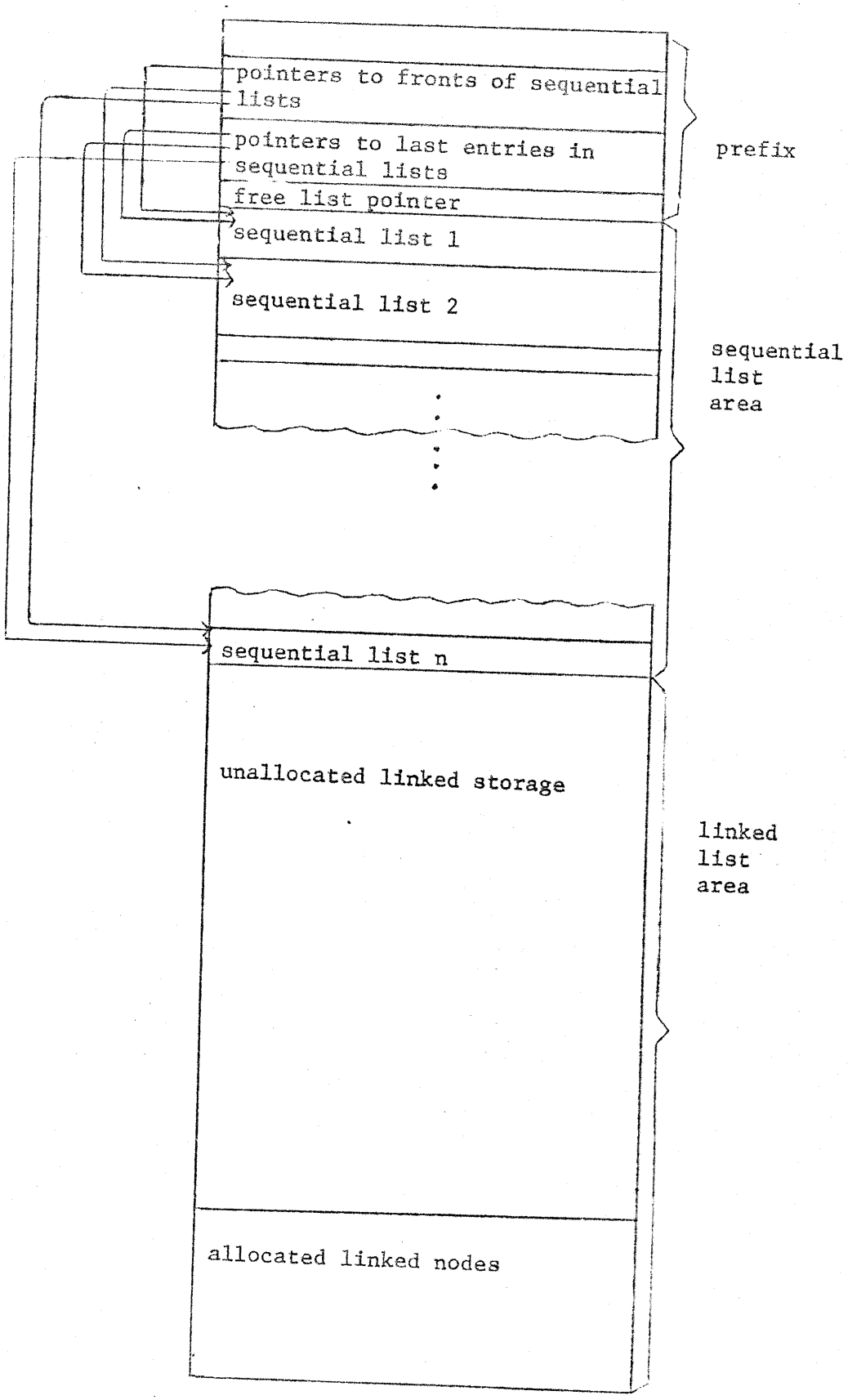

Figure 1: Data Base Area Storage Layout

self-descriptive. This makes it possible for the user to set up and access several data base areas simultaneously.

The sequential list area follows immediately after the prefix. The initial size of this area is determined by a global parameter. All sequential lists declared by the user through his initialization file will reside in this area. Space is initially allocated for each of these lists in a rather arbitrary fashion. Whenever any sequential list overflows its allocated area, an algorithm due to Garwick (see [1] pp. 242-246) is used to reallocate storage for all the sequential lists within the sequential area. There are three tables which are required by Garwick's algorithm. All are stored in the data base area prefix.

The linked list area extends from the end of the sequential area to the end of the data base area. All linked lists built by the user are stored here. Initially the entire linked area is considered to be available for use in creating linked nodes for linked lists. Whenever a request for the allocation of a linked node occurs, storage for the node is allocated from the end of the available linked area. Hence the linked lists grow from the end of the data base area towards the end of the sequential area. This facilitates the process of expanding either the linked or sequential area at the expense of the other, should such action become desirable. The free list pointer, a pointer to the last location of the linked area which is currently available for linked node allocation, is maintained in the prefix.

It is important to observe that the data base area need not be entirely contained in central memory. All accesses to individual fields in the linked and sequential areas are made through calls to memory access routines. For small data base areas, these routines

are simply routines for extracting words or part words from a central memory array. For large data base areas, these routines should be thought of as entries into a paged memory system, which is responsible for fetching into central memory and holding, those sections of a disk or drum resident data base area which are of current interest. (It seems prudent, however, that the prefix be central memory resident at all times.)

In recognition of both the possibility of paging and difficulties in attempting random access to linked lists, it was decided, moreover, that routines should be provided which transform a linked list residing in the data base into a sequential list residing outside the data base area. Using such routines, a linked list can be copied into a linear array supplied by the user and then accessed by using the sequential list routines designed for data base resident sequential lists. Part of the data base prefix is recreated in this linear array so that these routines can access the list successfully. Conversely, routines are provided so that a list can be built in a linear array outside the data base as a sequential list, and then converted to a linked list in the data base.

Initialization

The function of the Initialization Program is to read in a file embodying the user's specifications of names, formats, etc., and build the tables necessary for creating and accessing the data aggregates as specified.

This data file is begun with a line containing the level number of the data base description. This level number must be updated every time the format of the data base is changed in order to insure that the rollin/rollout programs will work properly. This will be discussed in detail in a later section of this paper.

The balance of the data file consists of node and field specification lines. There must be a line naming each node type to be created, and a line specifying each field of every node type.

A node type description line contains 1) the name to be associated with the node type, and 2) the organization -- sequential or linked (coded as T or N) to be used in forming lists from this node type. Since the name will eventually enter the system as a FORTRAN integer variable, it is expected that the name supplied on this line will consist of a letter I, J, K, L, M, and N followed by five or fewer letters or digits. It is also possible to place a C in column 11 of this line denoting that the following line consists entirely of a prose message concerning the node. This message will be printed with the initialization summary statistics and is intended to be an automated documentation aid.

Immediately following each node type description line and associated message (if any), the user must place one field description line for every field which is a component of the node. The field description line is identified as such by having the letter F placed in column 1, followed by the name of the field. As with node names,

this must be a legal ANSI FORTRAN variable name. In addition care must be taken to see that all field names are unique, and different from all node names. There must also be an indication of the type of the field. Under the current implementation four types are allowed -- integer (I), real (F), pointer (P), and alphanumeric (A). Care must be exercised in the selection of field names to assure that the implicit type associated by ANSI FORTRAN with the specified name is always integer. It should be noted that a node may consist of fields of differing types.

The range of values of the field may also be specified on the field description line if the field is of type integer or real. If the range is specified (by placing the minimum and maximum values on the line in appropriate columns), then whenever the system attempts to store a value into the field during execution, that value will be compared to the range specification. Out of bounds values will cause an error message to be produced. Range checking can be suppressed either by omitting maximum and minimum values, by specifying that a field is of type N (integer with no range checking) or X (real with no range checking), or by suppressing checking at run time (run time checking is discussed later). As with node description lines, a C placed in column 11 alerts the program to the existence of a message line immediately following.

The input file is terminated by a sentinel line in the form of a field specification line for a field whose name is END***.

Figure 2 shows a section of a data initialization file used to create a data base area which is designed to hold information about FORTRAN programs. Two sequential nodes, ISTMTB and ISYMTB are specified by T's in column 1. C's in column 11 indicate that comment lines follow. Hence we see that ISTMTB and ISYMTB are intended to be a statement table and a symbol table respectively.

Following each sequential node description line are specifications

```
    2
T ISTMTB  C
STATEMENT TABLE
F LINOST IC
LINE NUMBER OF FIRST LINE OF STATEMENT
F IBLKST IC
NUMBER OF THE BASIC BLOCK CONTAINING THIS LINE
F ITYPST IC
STATEMENT TYPE CODE
F IIVHST PC
POINTER TO THE LINKED LIST OF INPUT VARIABLES (CONSISTS OF IIVNOD'S)
F IOVHST PC
POINTER TO THE LINKED LIST OF OUTPUT VARIABLES (CONSISTS OF IOVNOD'S)
F PCBLST FC                                                    0.0      1.0
FRACTION OF CHARACTERS THAT ARE BLANKS IN THIS STATEMENT
F IEXHST P
T ISYMTB  C
SYMBOL TABLE
F ISNAME IC
CODED REPRESENTATION OF THE SYMBOL NAME
F ISDTYP AC
THE TYPE CODE FOR THE SYMBOL--I IS INTEGER, R IS REAL,.....
F ISNTYP I                    0
F ISNDIM PC                                  1000
POINTER TO A LINKED LIST OF DIMENSION INFORMATION
F ISDATV IC
NUMBER OF DATA STATEMENT INITIALIZING THIS VARIABLE (IF ANY)
F ISCOMN I
F ISEQUV P
F ISSTMT P
N IIVNOD  C
THE LINKED NODE TYPE USED FOR INPUT VARIABLES TO STATEMENTS
F IIVNDX IC
INDEX OF THE VARIABLE IN THE SYMBOL TABLE
F LNKFD1 PC
FIELD BY WHICH THIS LIST IS LINKED TOGETHER
F IIVTYP I
N IOVNOD
F IOVNDX I
F LNKFD2 P
F IOVTYP IC
OUTPUT CATEGORY OF THIS VARIABLE FOR THIS STATEMENT
F END***
```

Figure 2:

A sample data deck for the Initialization Program.

for the fields comprising the nodes. Thus we see, for example, that LINOST, IBLKST and ITYPST are integer fields of statement table nodes, while ISNAME, ISNTYP, ISDATV and ISCOMN are integer fields of symbol table nodes. Descriptions of some of these fields follow their definition cards (a C in column 11 is used to indicate a following description card). In addition notice that ISNTYP is specified to hold only integers between 0 and 1000. All other integer fields have no bounds specified, and thus will have no range checking. PCBLST is the only real valued field specified. It is a field that is a member of the statement table nodes and can take values between 0.0 and 1.0. ISDTYP is the only specified alphanumeric field. It is a field that is a member of the symbol table nodes.

The fields IIVHST, IOVHST, IEXHST, ISNDIM, ISEQUV, and ISSTMT are pointer fields. The first three belonging to statement table nodes, the last three to symbol table nodes. As pointer fields, these fields will contain pointers to nodes of linked lists. From the comment line following IIVHST it can be seen that the author intended this field to point to a node defined by the description of IIVNOD. This is merely a comment. No explicit linkage is created by the system. The user has the power and obligation to properly build linked lists and create pointers to them.

In this case, the author has indicated that he will use IIVHST fields to hold pointers to linked lists consisting of nodes of type IIVNOD. Definition cards for the nodes of type IIVNOD as well as nodes of type IOVNOD are found after the description lines for the two sequential lists. N's in column 1 indicate that IIVNOD and IOVNOD nodes will be used to build linked lists. As with sequential node description lines, each linked node description line is followed by a comment line (optional) and field description lines for the fields of the node.

Finally, the data file is terminated by the END*** sentinel line.

The allocation of individual nodes, and accessing of specific fields within the individual nodes are done by the user by means of a comprehensive library of FORTRAN callable routines. These routines accept as parameters the symbolic node and field names specified on the above described data base initialization file. The initialization program builds the tables needed to associate these names with actual data base area locations.

Specifically, the initialization program reads in the above described file and produces 1) the data required to initialize tables and variables in four labelled COMMON areas called NODETS, FIELDS, DBTABS, and ARPARS, and 2) the actual card images of the labelled COMMON statements whose variables and arrays are to be initialized. These are written onto the two mass storage files referred to earlier.

The labelled COMMON block NODETS is a list of simple variables each of which is the name of a node type named in the input deck. The initialization value of each such variable is merely the integer indicating its order of appearance in the COMMON block list.

The labelled COMMON block FIELDS is likewise a list of simple variables, but each of these is the name of a field specified in the input deck. These variables too are initialized to integers indicating order of appearance.

Clearly, the process of creating and writing the FORTRAN line images declaring the labelled COMMON blocks FIELDS and NODETS is a simple matter. The COMMON statements generated for the blocks FIELDS and NODETS using the data file shown in Figure 2 is shown in Figure 3. Once these line images are on the mass storage file, they must be punched

```
      COMMON/NODETS/ISTMTB, ISYMTB, IIVNOD, IOVNOD
      COMMON/FIELDS/LINOST, IBLKST, ITYPST, IIVHST, IOVHST, PCBLST, IEXHST,
     *ISNAME, ISDTYP, ISNTYP, ISNDIM, ISDATV, ISCOMN, ISEQUV, ISSTMT, IIVNDX,
     *LNKFD1, IIVTYP, IOVNDX, LNKFD2, IOVTYP
```

Figure 3

The COMMON statements declaring COMMON blocks NODETS and FIELDS written to INCDAT after execution of the initialization program using the deck shown in Figure 2 as input.

out and inserted manually into a user's program decks, or written directly into his program files either by a FORTRAN preprocessor or by a system routine (such as INCLUDE under EXEC 8 on the UNIVAC 1108). In any case, this is the mechanism by which list and field names, as specified by the user in his specification file, become meaningful variable names, usable from his FORTRAN code.

The initialization values of the variables in these two blocks, are written to a binary file during this initialization run and then set as values of the proper variables during the execution of a startup subroutine which must be invoked before the user attempts to use any of the routines in the data accessing library.

The labelled COMMON blocks DBTABS and ARPARS contain the variables and arrays which are used to associate field and node names with the corresponding offset positions in data base areas. These arrays and variables are also used to check the correctness of access requests. The statements declaring these COMMON blocks are created by the initialization program and written out to the mass storage file of symbolic line images. They are later incorporated into the source text of the routines of the data accessing library by the preprocessor or INCLUDE facility. The values of the variables in these COMMON blocks are also created here. They are written to the binary mass storage file by this program and read from the file into the corresponding variables during execution of the startup subroutine mentioned above.

A complete description of the tables and variables contained in DBTABS and ARPARS can be found in [2], along with detailed descriptions of how they are created by the initialization program and used by the routines of the data accessing library.

In an effort to use storage efficiently the system allows for the

creation of fields of two different sizes -- full word and part word. In the current implementation pointer fields occupy a part word, while alphanumeric fields, real fields, and integer fields without range checking occupy full words. In the case of integer fields with range checking specified, a computation is made to determine the most appropriate field size. Two implementation parameters which must be supplied through a BLOCK DATA subprogram enable the initialization program to make this determination. These parameters are NUMBTS, the word size of the machine in bits, and NPWPFW, the number of part word fields which are to be placed into a full word for this implementation. Using these two parameters the maximum integer size accommodated by a part word field is easily determined. Then, if an integer field has range checking specified and the indicated range of integers can be accommodated by a part word field, the system allocates only a part word field for it.

It is worthwhile to note in closing here that the initialization program also prints out a summary of its activities. All node type names which have been declared are printed out. The fields comprising each node type are also named and described with respect to data type, range, and location within their node.

Any descriptive comments inserted by the user in his input deck are printed out next to the items he wished to have described. The total numbers of sequential node types, linked node types, and fields are listed in a summary table which also lists the sizes in number of words and number of fields for all nodes. Finally the program produces dumps of both the symbolic and binary mass storage files. In this way the system helps in the creation of documentation for the user's programs.

An Example of the Use of the Data Accessing Library

As mentioned before, it was decided that a body of powerful, high-level data base manipulation routines would not be built. Instead a comprehensive library of low level routines was created. These routines perform such tasks as seizing and initializing new nodes for the various lists, entering and retrieving data into and from specified fields, and searching lists for prespecified data items. It is expected that higher level subroutines will easily be constructed from these lower level routines in response to future needs. The names of the routines comprising this data accessing library are listed in Appendix A along with their calling sequences and brief descriptions of what they do.

As an example of the operation of the data base system, Figure 4 shows a sequence of FORTRAN code which operates on a data base area created according to the specifications in Figure 2. Figures 5a, 5b and 5c show an example of how the code in Figure 4 is used to transform an existing data base area. As noted in the comments for subroutine ENTIVS, the code creates a statement table node and a linked list of the input variables to the statement. Specifically, in Figure 5c we see that 1) the L^{th} node of table ISTMTB is created and added to ISTMTB, 2) a statement type code of 15 (as dictated by the input data) is inserted into it, and finally, 3) an input variable list of four nodes is created in the linked area and appended to the new ISTMTB node, indicating that the symbols entered into ISYMTB as items 6, 4, 2, and 1 respectively are inputs to statement L. It should be noted that ENTIVS uses six separate routines (NXTPOS, PUTTBL, ITBSCH, NEWNOD, PUTLST, and ADLSTT) in the data base accessing library to perform these operations.

```
      SUBROUTINE ENTIVS(NTP, NIVARS, IVCODS, IAREA)
C  THIS SUBROUTINE CREATES A STATEMENT NODE FOR A NEW STATEMENT
C  IN THE DATA BASE AREA IAREA.  IT
C  SETS THE TYPE FIELD OF THE NODE TO NTP.  IT THEN
C  SEIZES AN INPUT VARIABLE NODE FOR EACH INPUT VARIABLE
C  IN THE STATEMENT.  THE NUMBER OF SUCH VARIABLES IS
C  PASSED AS THE VALUE OF NIVARS, AND REPRESENTATIONS OF
C  THE VARIABLE NAMES ARE PASSED AS INTEGERS IN THE ARRAY
C  IVCODS.  THE INPUT VARIABLE NODES ARE SET TO CONTAIN
C  THE LOCATIONS IN THE SYMBOL TABLE OF THE APPROPRIATE
C  SYMBOLS AND ARE LINKED ONTO THE LIST OF INPUT SYMBOLS
C  FOR THIS STATEMENT.  THIS LIST IS LINKED OUT OF THE IIVHST
C  FIELD OF THE NEW STATEMENT NODE.
      COMMON/NODETS/ISTMTB, ISYMTB, IIVNOD, IOVNOD
      COMMON/FIELDS/LINOST, IBLKST, ITYPST, IIVHST, IOVHST, PCBLST, IEXHST,
     *ISNAME, ISDTYP, ISNTYP, ISNDIM, ISDATV, ISCOMN, ISEQUV, ISSTMT, IIVNDX,
     *LNKFD1, IIVTYP, IOVNDX, LNKFD2, IOVTYP
      DIMENSION IAREA(1), IVCODS(1)
C  SEIZE A NEW STATEMENT TABLE NODE--ITS NUMBER IS L
      L = NXTPOS(ISTMTB, IAREA)
C  INSERT TYPE CODE
      CALL PUTTBL(NTP, ITYPST, L, ISTMTB, IAREA)
      N = 1
C  LOOP FOR ADDING INPUT VARIABLES ONE AT A TIME
10    IF(N.GT. NIVARS) RETURN
```

```
C  FIND LOCATION IN ISYMTB OF SYMBOL REPRESENTED BY
C  IVCODS(N).  IF NOT FOUND PRINT ERROR MESSAGE
      ISYLOC = ITBSCH(IVCODS(N), ISNAME, ISYMTB, IAREA)
      IF(ISYLOC.NE.0)GOTO30
      WRITE(6,20)IVCODS(N)
   20 FORMAT(15H ERROR--SYMBOL , I10, 13H NOT IN TABLE)
      GOTO 40
C  SEIZE NEW LINKED NODE, LOAD SYMBOL REPRESENTATION
C  INTO IT AND LINK IT INTO INPUT VARIABLE LIST
   30 LOC = NEWNOD(IIVNOD, IAREA)
      CALL PUTLST(ISYLOC, IIVNDX, IIVNOD, LOC, IAREA)
      CALL ADLSTT(IIVHST, L, ISTMTB, LNKFD1, IIVNOD, LOC, IAREA)
   40 N = N + 1
      GOTO 10
      END
```

Figure 4

A subroutine using data base manipulation routines.

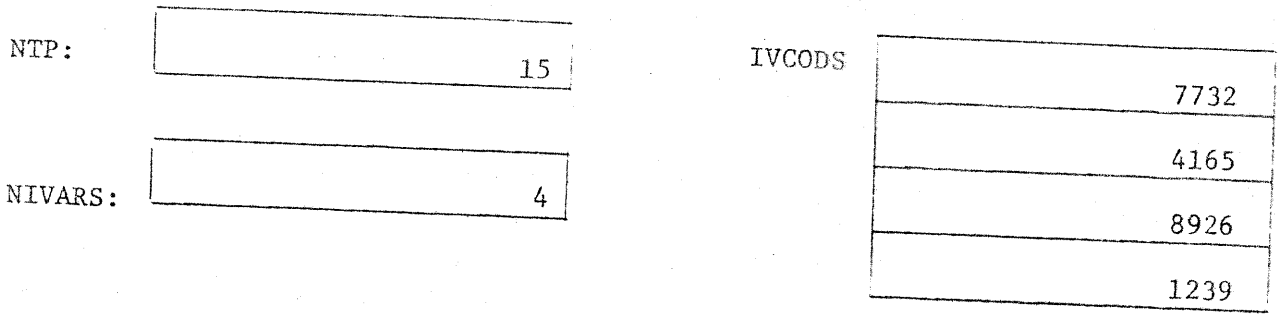

Figure 5a: Sample input to the routine shown in Figure 4.

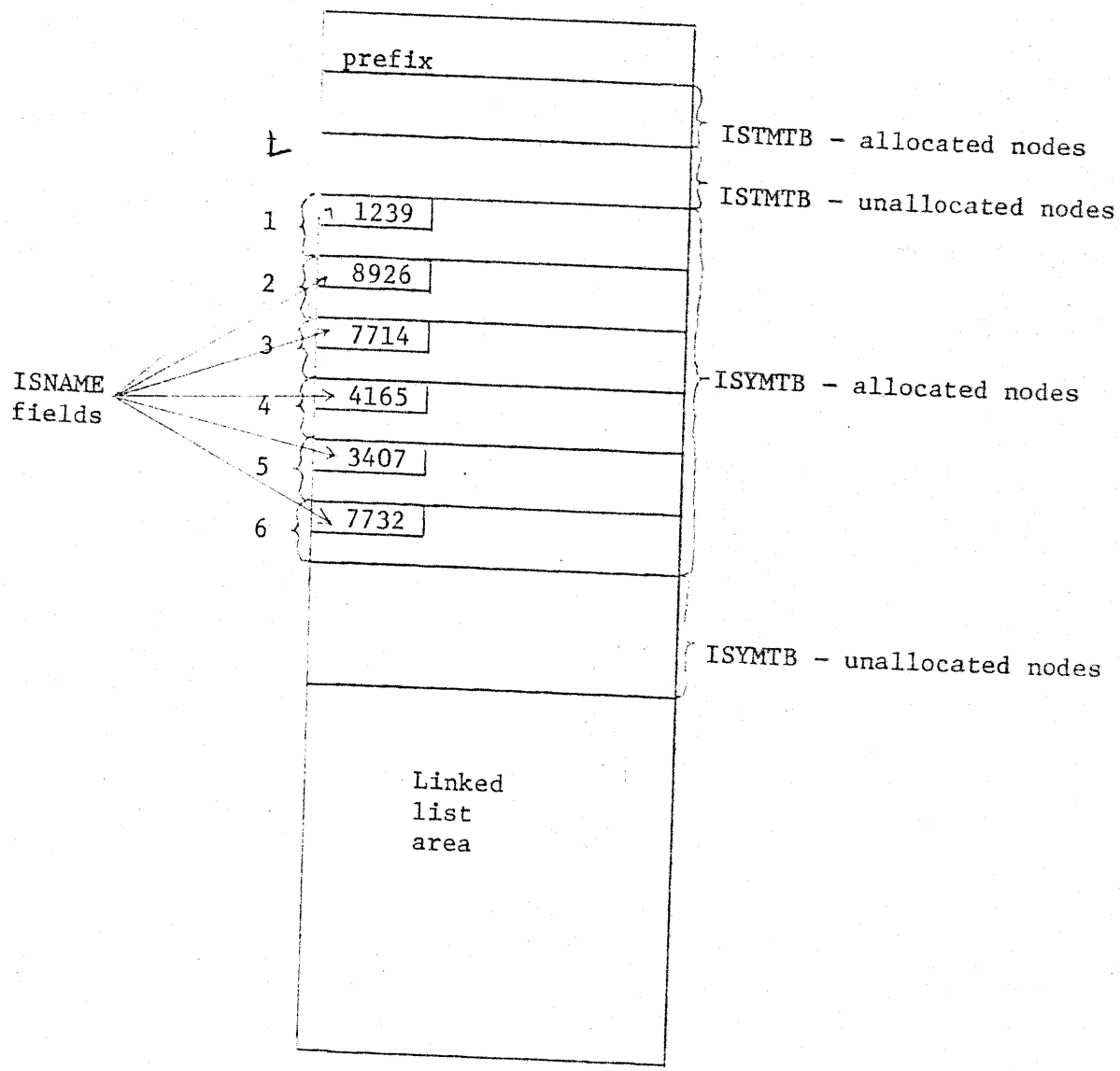

Figure 5b: A portion of data base area IAREA before execution of the subroutine shown in Figure 4.

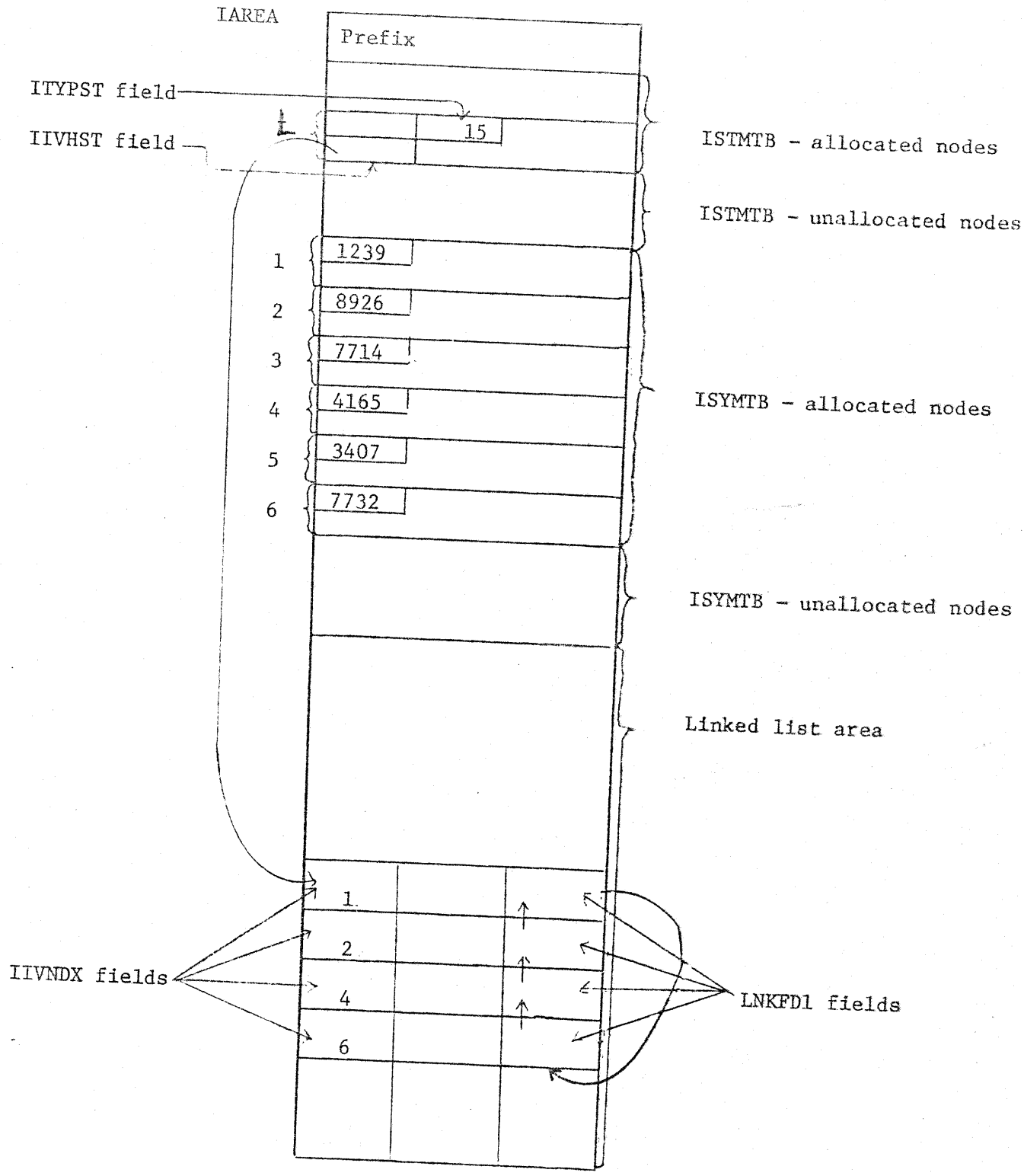

Figure 5c: A portion of data base IAREA after execution of the subroutine shown in Figure 4 using the data shown in Figure 5a.

The workings of these and most of the other data accessing routines are described briefly in appendix A and in detail in [2].

A brief description of the handling of linked lists seems in order here. A linked list or treelike structure consisting of virtually any configuration of linked nodes could readily be constructed by using just the low level node creation and field accessing routines of the data accessing library. It was felt, however, that the linked list structure most often created would be a list consisting entirely of nodes of the same type. For this reason a set of routines designed to create, augment, and search such lists was built and imbedded in the library (ADLSTT is a member of this set) as a convenience for the user.

The linked lists that are created and maintained by these routines are circularly linked. The list header, which must be a pointer field in a sequential or linked node, points to the last node on the list. The last node points to the first node on the list, which in turn points to the second node on the list and so forth. This structure was selected because it facilitates the accessing and addition of nodes at either end of the list.

Error Detection

In designing the data base system heavy emphasis was put on detecting user errors. The general philosophy employed was to allow the user to obtain as much error feedback as possible, as soon as possible, in order to help him avoid cascading errors. It was felt that checking for a wide variety of possible error conditions would greatly simplify the process of debugging a program, and also help to acquaint a new user with the data base system itself.

In placing error checking code throughout the system we attempted to anticipate a wide range of possible errors. For example, argument checking is done for virtually all data accessing routines. Such routines invariably expect as arguments a field name, node type name, individual node specification and data base area vector. The checking routines attempt to verify that the vector passed is in fact a data base area. They verify whether or not the individual node specification actually specifies a node which has previously been allocated (this is possible because all such allocations must be done by means of system routines). They verify whether the type of the node (linked or sequential) to be accessed agrees with the type of the access routine invoked (the routines used to access linked nodes are different from those used to access sequential nodes). They also verify that the field name passed is actually a field of the node whose name has been passed. Other types of error checking are also present.

Bounds checking is performed. In general, before any routines in the data accessing library attempt to access an individual word in the data base area, the address of the word is examined to verify that the word actually lies within the data base area. In addition,

it is possible to determine whether the word to be accessed should lie in either the sequential area or the linked area. In such cases a check is made to determine whether the computed address is in the correct part of the data base area. In addition, the whole concept of data item range checking is designed to provide the user with an early warning that the values he is creating and storing do not conform to his prestated estimates. Thus the range checking feature is very much in keeping with our objective of comprehensive error detection.

Clearly, it is our feeling that references to subscripts that are out of array bounds are a major source of FORTRAN errors. In a linked structure, moreover, such references can be most easily made and particularly disastrous. In order to protect the user from this kind of error and to encourage the user to allow the system to do pointer manipulation, all pointer values are flagged with a readily distinguishable bit configuration by the system making it possible to examine an arbitrary value for the presence of pointer flag bits. The implementor must define in a block data subprogram a COMMON variable which contains this pointer bit configuration. Those routines in the data accessing library which require pointers as arguments examine the values passed to them to verify that values which should be pointers are in fact pointers.

In a similar vein, the system also recognizes a special empty bit - configuration. It is often useful to be able to determine whether a particular field has been initialized with a value or not. For this reason, all nodes, during their initial allocation, have all of their fields set to empty by the allocation routines. Because the system recognizes both full word fields and part word fields, there are two different empty bit configurations -- full word empty and part word

empty. (Both must be defined by the implementor by means of COMMON variables in a BLOCK DATA subprogram.) Because an attempt to read an empty field out of a data base is often symptomatic of an error, the system prints out a warning message whenever a user attempts to do so.

In designing the error handling mechanisms for the system, a careful attempt was made at producing meaningful error output. Thus, most error messages are accompanied by lists of pertinent parameters. In addition, in the case of erroneous field references, the actual names of the offending node types and fields are printed. The node type and field references are passed to the checking routines as integer values, derived from the variables in the COMMON blocks FIELDS and NODETS. These values would be of little meaning to the user, however. Hence they are used as indices into arrays containing the alphanumeric representation of the node type and field names, which are presumably more meaningful. It is these alphanumeric names that are printed out for the user by the error checking routines.

It is felt that these error checking facilities provide the user with tools for detecting a wide range of errors at their source. This is a powerful tool in being able to rapidly and accurately isolate faulty usage of the data base system. Checking, however, can be costly in execution time when employed in a well tested, operational program. Therefore, all data accessing routines give the user the option of either selecting or suppressing checking. A COMMON logical variable determines whether checking is done. The value of this variable can be changed within a program, thereby allowing the user to control the segments of his code that are to be tested.

This two level (all or none) approach to checking seems to be a desirable feature, but it should be pointed out that additional

levels would enhance the system's flexibility. For example it would probably be useful to have an optional level of error checking solely for detecting "empty" field retrievals. The "empty" bit configuration discussed earlier enables us to make this test. Often, the retrieval of such "empty" fields is quite reasonable and is not indicative of an error. In such cases the user probably does not want to have a warning message generated. Under the current implementation he can suppress the warning only by suppressing all error checking -- an unreasonably harsh alternative. Under a multilevel error detection scheme, this type of error checking could be selectively enabled and disabled independently of other error checking.

Interprogram Communication of Data Base Areas

The purpose of creating a data base is to store an existing body of data for future reference. The data base accessing system described so far will readily allow a program to reference a data base which has been created during the execution of that program. Often, however, it is useful for a program to be able to reference a data base which has been set up during the execution of a previous program. If the data base is central memory resident, the usual procedure is to have the first program generate the data base and write it out to a mass storage device such as a disk or tape, and then have the second program begin its execution by reading in the data base from mass storage. The second program is then subsequently able to access the data base.

We have created a pair of routines, ROLLIN and ROLOUT, for storing and recalling central memory resident data bases as described above. Such routines are easy to write assuming that the data base description has not been changed between the execution of the run which creates and stores the data base and the execution of the run which recalls and uses the data base. In particular, the first program must call ROLOUT, naming as parameters the data base area to be saved and the mass storage file on which to save the data base. ROLOUT then writes out the total number of words in the data base area, followed by the entire data base area, using a binary write. (Note that the total size of the data base is readily available, being stored in the prefix.)

The second program can then recall the data base area by a call to ROLLIN. A description of ROLLIN and its parameters can be found in Appendix A. ROLLIN must read in the first word of the mass storage file holding the data base and use it as a count of the number of succeeding words to be read in to restore the data base. Assuming

the data base description has not changed between the two runs, the data base can now be accessed as described in the earlier parts of this paper. If the data base description has been altered in the interim, however, the ROLOUT/ROLLIN scheme described above will not work. In general, the node type and field description vectors in DBTABS will have changed between the ROLOUT and ROLLIN. Relative locations of fields within nodes will have changed, and the use of new accessing vectors on old data base areas will result in disaster.

This problem is very real and important for this system because an overriding design goal was to allow for flexibility and ease of alteration of the data base description. Thus it is ordinarily expected that modifications to the data base description might be made fairly regularly. It is not unreasonable, however, to envision the following situation. A data base is created at sizeable cost. It retains its validity throughout a series of relatively minor data base description modifications, and it still is required by a particular program over a period of several weeks or months during which the data base format is changing. It may be costly to regenerate the data base anew in accordance with each new data base description. It is far preferable to create a more sophisticated ROLOUT/ROLLIN package which automatically detects alterations in data base formats and rolls in an old data base in the current format so that it can be used immediately with current accessing vectors.

Under such a scheme a user need never even know that the data base description has changed (unless of course, fields or node types which his program references have been deleted). He simply rolls in the data base and references node types and fields via the symbolic names which he has always used. This section describes such an adaptive,

dynamically reformatting ROLOUT/ROLLIN package.

The ROLOUT algorithm is far more simple than the ROLLIN algorithm. In order to save a data base area ROLOUT copies out the data base as well as the data base description (access information) onto the mass storage file.

The more difficult aspects of storing a data base arise upon trying to ROLLIN the data base. ROLLIN has been organized into six major subroutines: READIN, COMPAR, GARBGE, ADJPTR, CPYOUT, and COPYIN. Figure 6 is a diagram of the flow of control through the routines of the ROLLIN process.

The first routine, READIN, begins by reading in the stored data base and the old accessing vectors from the designated mass storage file. At this point, the routine detects if there has been a change in data base description by comparing the old vectors with the current vectors, previously initialized by the usual data base routine for reading the vectors off of their mass storage file. If there have been no changes in the description, then the rest of the routines are skipped and ROLLIN returns. It is possible for the user to force execution of these routines, however, in order to obtain certain desirable side effects of the reformatting process. These side effects will be discussed at a later point in this section.

The second phase of the ROLLIN process involves comparing the descriptions of the two (presumably different) data bases to decide what data must be transferred into the new data base. The routine that does this comparison is named COMPAR. When COMPAR is called to compare the descriptions of the data bases, it checks to find which node type and field names are found in both descriptions. For these node types and fields, COMPAR forms a mapping function from one description into the other description.

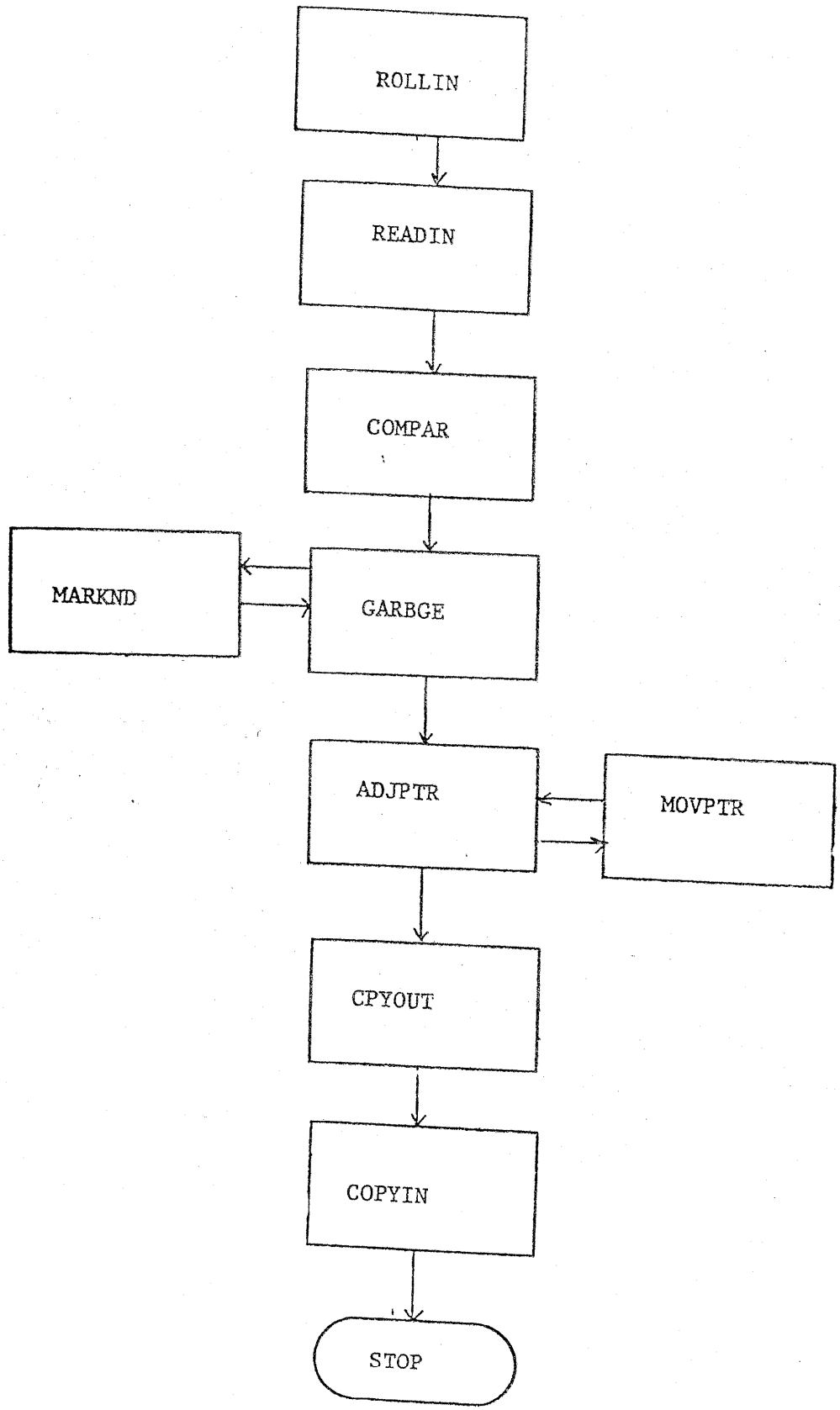

Figure 6: Diagram representing the flow of control through the routines of the ROLLIN process.

The third routine of ROLLIN is called GARBGE. To make sure that the new data base will have only the nodes desired, some nodes may be eliminated from the data base. A basic assumption of the data base system is that all meaningful information in the linked area is accessible from some pointer field in a sequential list or some chain of pointers originating in a pointer field of a sequential list. To find these points GARBGE steps through the sequential lists that transfer into the new data base and inspects within every such sequential list, every pointer field that transfers. If a pointer field is found to be non-empty, then the pointer is to be followed into the linked area. To follow the pointer, GARBGE calls the routine MARKND.

MARKND is an adaptation of a classical garbage collection marking scheme (see Knuth [1] for a thorough treatment of garbage collection marking algorithms). Upon reaching a linked node, MARKND examines the destination address field and the visited field to determine whether the node has already been reached during execution of GARBGE. If so, one of the fields will have been set to a nonzero value, and MARKND does nothing further with the node. If not, MARKND examines the node to determine whether the type of the node is named in the new data base description. If not, a "visited" mark is placed in the node and nothing further is done with this node. If so, then the destination address field of the node is loaded with a destination address. This is the offset within the new data base area at which the node will eventually be loaded. This offset is determined while GARBGE is executing in the following way. During an initialization phase of GARBGE an offset pointer is initialized to the last location in the new data base area (this must be an input parameter to ROLLIN).

Whenever MARKND encounters a node which belongs in the new data base area, the node's size is determined and the starting address of the node is determined by subtracting the size from the current offset pointer value. The result is placed into the destination address field of the node. The offset pointer is then updated to this value.

Having thus processed the current node, MARKND then proceeds to mark all nodes reachable from it via pointer fields in the node. All but one of these pointers are pushed on a stack, and MARKND processes the other exactly as described above. When a node and all of its successors have been completely processed, MARKND pops a new node off of its stack and continues. Eventually the stack becomes empty, and MARKND returns to GARBGE, having marked all linked nodes reachable directly or indirectly through the pointer field originally passed by GARBGE. When GARBGE has invoked MARKND for all pointers from the sequential area into the linked area, then all nodes accessible in the new data base will have an address in their destination address field. If the destination address field of a linked node does not have an address, then that linked node need not be transferred into the new data base.

When GARBGE has completed marking nodes with their forwarding addresses, the ADJPTR routine is entered. ADJPTR changes all pointer fields to point to node locations within the new data base. ADJPTR steps through the sequential lists looking for pointers. When a pointer is found, the pointer field is adjusted to point to the address contained in the destination address field of the node pointed to, provided that the field contains an address. If the field does not contain an address, then the node does not transfer into the new data

base and so the pointer field is set to "empty". Once the sequential pointer is adjusted, then all pointers within linked nodes reachable from the sequential pointer must be changed. ADJPTR calls MOVPTR to change these pointers. MOVPTR works much like MARKND, stacking, unstacking, and resetting pointers. If a node has been eliminated, then all pointers to the node are set to "empty" and the pointers from the node are not followed. The mark and destination address fields are used to insure that a node is not visited twice.

When all of the pointers to linked nodes have been adjusted, all that remains is to move the nodes into their new locations. We have devised inplace algorithms for doing this, but all are awkward and inefficient. Hence in order to move the nodes to their new locations, we write the nodes that transfer to the new data base onto an auxiliary mass storage file and then read them back in at their new address locations. The routines to do this copying are called CPYOUT and COPYIN. CPYOUT copies the sequential lists that transfer and then steps through the linked area copying out only the linked nodes that are marked with a destination address. COPYIN does the complicated job of reading the nodes back in and transforming the old nodes into the new nodes. By using the tables generated by COMPAR, as well as both the old and new data base accessing vectors, COPYIN can transform the nodes. To transform a node, COPYIN must extract data items from the old nodes, insert them into the new nodes in the proper offset positions (but only for those fields which transfer), set any new fields in the nodes to "empty", and reset the mark and destination address fields of all linked nodes to zero.

It is worthwhile to note here that it is possible for the description of "empty" or the pointer flag to change between the time data base was rolled out and the time it is rolled in again. If either has changed, then COMPAR makes note of the fact and CPYOUT/COPYIN recognizes the old empty or pointer flag in the old data base, while writing the new empty or pointer flag into the new data base.

As alluded to earlier, some interesting side effects of the ROLOUT/ROLLIN process may make the use of all the phases of ROLLIN desirable. For example, ROLLIN eliminates any linked nodes not pointed to directly or indirectly from the sequential tables in much the same way that a garbage collection would, thereby making more room for new data aggregates. Another advantage of the complete ROLLIN operation is that it tends to localize linked lists into a smaller address space. This localization becomes a distinct advantage when the data base is accessed through a paging scheme, because all of the linked list nodes comprising a given list will be stored closer together after ROLLIN. Hence the list will cross fewer page boundaries. Thus accessing an entire linked list should involve fewer page faults. To allow a user to obtain these desirable side effects even though there has been no change in the data base description, a logical parameter was included in the call to ROLLIN. If the parameter is .TRUE. then the full six steps of ROLLIN will be taken regardless of whether or not the data base description has changed between ROLOUT and ROLLIN.

Conclusions

This system is embodied in a collection of approximately 75 FORTRAN program units consisting of approximately 4000 source statements. Approximately 10 man-months of effort were required for the design, coding, checkout and documentation of the system. The system has been in operation for approximately 18 months on the CDC 6400 at the University of Colorado and it has been heavily used for nearly 12 months by the FORTRAN program validation project at the University of Colorado, which provided the original impetus for its creation. During this time the system has been successfully transported to a similar machine (the CDC 7600) operating under a different operating system.

This experience provides an interesting case history which we believe indicates that the system has successfully met its design goals. The FORTRAN validation project has to date produced in excess of 20,000 FORTRAN source statements comprising a system which analyzes FORTRAN programs. At the outset the project investigators were unsure about the nature of the analytic procedures which would be desirable in validating programs. Hence tentative decisions were made about the design of these procedures and the nature and organization of the data which they would require. As the project progressed the structure and quantity of the needed data was expected to change often. Thus a list structured data base was set up using the data base system described here. The tentative data aggregates were defined by means of an initialization run, and programs accessing the data aggregates thereby defined were immediately produced. The expectation of the project investigators was that these programs would never have to be

altered because of changes in the structure of the data aggregates. As time progressed and such changes to data aggregate organization were made, this expectation was borne out. Data aggregate organization was altered frequently, sometimes drastically, without any necessity for reprogramming due to these changes. As a result the investigators were able to concentrate on the design of their validation system, free of worry about possible reprogramming due to changing requirements for data items and organizations.

The error detection capabilities of the system proved useful, consistently allowing programmers to pinpoint errors close to their source. The ROLLIN/ROLOUT capability also proved useful, but in a somewhat unexpected way. After several months, work was begun on a new analytic system requiring only some of the data needed and created by the old validation system and requiring the derivation of certain new data aggregates. After the creation of a new, parallel data base description for this new project it became possible to use the ROLLIN/ROLOUT routines to transform data bases created by the original analytic system into data bases usable by the new system. In this way, the high cost of recreating the data bases was traded off for the lesser cost of simply reformatting them. Hence it proved feasible and useful to create a given data base once and have it shared, through reformatting, by two different analytic systems.

Although the data base system has proven to give the desired flexibility, certain drawbacks have become apparent. The inefficiencies in time and storage, expected in a system such as this, have definitely been noticed. The data base system makes only a perfunctory attempt at packing data items into words efficiently, thereby reducing the effective size of data bases. In addition, each data access must be made through a long chain of subprogram invocations at high cost in

execution speed. (Execution time can be speeded up somewhat, however, by suppressing most diagnostic checks as described above).

Any change in a data base description generally entails the recompilation of all subprograms which refer to the data base, invariably an expensive process.

In addition, a program which makes heavy use of the data base system inevitably takes on an unreadable, un-FORTRAN-like appearance, due to the profusion of subprogram invocations. There appears to be no alternative to this heavy use of subprograms which does not involve an extensive preprocessing scheme or extensions to the FORTRAN language itself. Both schemes were rejected by our group as entailing more effort and allowing less flexibility than the scheme described here. It is worth observing that part of the unreadability of the programs using our system, however, is attributable to the ANSI FORTRAN limitation on variable and subprogram names to six or fewer characters. We believe that by allowing longer names, the readability of these programs, and indeed all FORTRAN programs could be greatly enhanced. Carrying this last point to its logical conclusion, it seem clear that FORTRAN is basically not a very suitable language at all for producing systems which must deal with complex or changeable data aggregates. Non-trivial data aggregates are not provided for adequately in ANSI FORTRAN, thus any attempt at providing for them seems destined to appear artificial and contrived. Many newer languages (e.g., PL/I and PASCAL) provide more adequate data aggregate capabilities. Unfortunately, at present these languages are less well-established than FORTRAN and programs written in them are far less portable. Hence the system designer is faced with a choice between probable loss of portability and probable loss of clarity and esthetic appeal. Often, as in the

case of the validation project, the need for portability must prevail.

Having weighed the various factors, we have concluded that the above cited weaknesses clearly reduce the usefulness of our system in producing production programs, but do not outweigh its advantages in building prototypes. Loss of efficiency, relative opacity of source code and frequent recompilations are the usual prices one pays in any large software development effort. The program validation group was not spared these costs by using the data base system. They were spared worries about description of plans and schedules due to changes in data organization. In addition, because the data base system is FORTRAN based and highly portable, the prototype which uses it also has good portability characteristics.

In summary, we believe that this case history gives a good indication that the data base system is a valuable tool for use in constructing prototype systems where the use of FORTRAN is dictated by other considerations, such as portability.

REFERENCES

1. D. E. Knuth, "The Art of Computer Programming", V. 1, Fundamental Algorithms, Addison Wesley, Reading, Mass., 1969.

2. L. Osterweil, L. Clarke and D. W. Smith, "A FORTRAN System for Flexible Creation and Accessing of Data Bases". Department of Computer Science, University of Colorado, Report #CU-CS-052-74, August 1974.

Appendix A

List of Subprograms in the Data Accessing Library

User level data accessing routines

ADLSTL(IFIELD,NDTYPH,LOCH,LNKFLD,NODTYP,LOC,IAREA)

ADLSTL is a subroutine that adds a node to the end of a linked list. The list header is contained in a field of a linked list node. The lists maintained by this routine are circularly linked.

ADLSTT(IFIELD,NUMBER,ITABLE,LNKFLD,NODTYP,LOC,IAREA)

ADLSTT is a subroutine that adds a node to the end of a linked list. The list header is contained in a field of a sequential list node. The lists maintained by this routine are circularly linked.

CPLIST(LNKFLD,NODTYP,LOC,IAREA,IBUF, IBUFSZ)

CPLIST is a subroutine that copies a circularly linked linear list in a data base array into a sequential list in a linear array. The list can then be accessed by the sequential list accessing routines.

INITDB(IAREA,IASIZE)

INITDB is a subroutine that initializes an array as a data base. Before an array can be used as a data base this subroutine must be called.

INITRN(INPFIL)

INITRN is a subroutine to initialize the common blocks used in the data base routines. This subroutine must be called before any data base routines are executed.

ISMPTY(IWORD,IFIELD)

ISMPTY is a logical function that returns the value true if IWORD is flagged as an empty field and returns the value false otherwise. This is a machine dependent routine.

ITBSCH(INFO,IFIELD,ITABLE,IAREA)

ITBSCH is a function that searches a designated field of every node of a sequential list to determine if a designated data item is present. ITBSCH returns as its value the sequence number of the first node to contain the data in the specified field. If the data is not found ITBSCH returns a value of zero.

ITMLST(IFIELD,NODTYP,LOC,IAREA)

ITMLST is a function that returns as its value the contents of a field of a linked list node. There exists a corresponding real valued function XTMLST for fields that contain floating point values.

ITMTBL(IFIELD,NUMBER,ITABLE,IAREA)

ITMTBL is a function that returns as its value the contents of a field of a sequential list node. There exists a corresponding real valued function XTMTBL for fields that contain floating point values.

LSTPOS(ITABLE,IAREA)

LSTPOS is a function that returns as its value the sequence number of the last node allocated to a sequential list.

LSTSCH(INFO,IFIELD,NODTYP,LOC,LNKFLD,IAREA)

LSTSCH is a function that searches a designated field of every node of a linked list to determine if a designated data item is present.

LSTSCH returns as its value a pointer to the first node on the list to contain the data in the specified field. If the data is not found in the linked list nodes then LSTSCH returns a value of zero. LSTSCH requires that the designated linked list be circularly linked.

MKLSTL(IFIELD,NDTYPH,LOCH,LNKFLD,NODTYP,IAREA,IBUF)

MKLSTL is a subroutine that copies and transforms a sequential list stored in a linear array into a linked list stored in a data base array. The list header is in a field in a linked list node. If the header already points to a linked list consisting of nodes of the same type as the list to be created then the old list is overwritten.

MKLSTT(IFIELD,NUMBER,ITABLE,LNKFLD,NODTYP,IAREA,IBUF)

MKLSTT is a subroutine that copies and transforms a sequential list stored in a linear array into a linked list stored in a data base array. The list header is in a field in a sequential list. If the header already points to a linked list consisting of nodes of the same type as the list to be created then the old list is overwritten.

MTYLST(IFIELD,NODTYP,LOC,IAREA)

MTYLST is a subroutine that flags a field of a linked list node as empty. Note that system routines flag all fields of a newly allocated node as empty.

MTYTBL(IFIELD,NUMBER,ITABLE,IAREA)

MTYTBL is a subroutine that flags a field of a sequential list node as empty. Note that the system routines flag all fields of a newly allocated node as empty.

NEWNOD(NODTYP,IAREA)

NEWNOD is a function that allocates a linked list node from the free list and returns as its value a pointer to the allocated node. NEWNOD sets all fields of the node to empty.

NXTPOS(ITABLE,IAREA)

NXTPOS is a function that allocates a new node to a sequential list. NXTPOS returns as its value the sequence number of the new node. NXTPOS sets all fields of the node to empty.

PUTLST(INFO,IFIELD,NODTYP,LOC,IAREA)

PUTLST is a subroutine that enters data into a field of a linked list node. This subroutine enters only data of type integer. There exists a corresponding subroutine XPTLST to enter floating point data.

PUTTBL(INFO,IFIELD,NUMBER,ITABLE,IAREA)

PUTTBL is a subroutine that enters data into a field of a sequential list node. This subroutine enters only data of type integer. There exists a corresponding subroutine XPTTBL to enter floating point data.

ROLLIN(IAREA,IASIZE,IFLNM,IAUX,IGCFLG)

ROLLIN is a subroutine that reads in a data base array and node and field descriptions from a file. If the current node and field descriptions have been modified, the data base is transformed into the current data base format. Garbage collection occurs whenever the data base must be transformed or upon request.

ROLOUT(IAREA,IFLNM)

ROLOUT is a subroutine that writes an entire data base array and node and field description onto secondary storage for later use.

STLIST(NODTYP,IBUF,IBUFSZ)

STLIST is a subroutine that initializes a linear array so that a linked list may be created and accessed as a sequential list within the vector.

TOPLSL(IFIELD,NDTYPH,LOCH,LNKFLD,NODTYP,LOC,IAREA)

TOPLSL is a subroutine that adds a node to the front of a linked list. The list header is contained in a field of a linked list node. The lists maintained by this routine are circularly linked.

TOPLST(IFIELD,NUMBER,ITABLE,LNKFLD,NODTYP,LOC,IAREA)

TOPLST is a subroutine that adds a node to the front of a linked list. The list header is contained in a field of a sequential list. The lists maintained by this routine are circularly linked.

XPTLST(XINFO,IFIELD,NODTYP,LOC,IAREA)

XPTLST is a subroutine that enters a real data item into a field of a linked list node. The corresponding subroutine for integer data is PUTLST.

XPTTBL(XINFO,IFIELD,NUMBER,ITABLE,IAREA)

XPTTBL is a subroutine that enters a real data item into a field of a sequential list node. The corresponding subroutine for integer data is PUTTBL.

XTMLST(IFIELD,NODTYP,LOC,IAREA)

XTMLST is a function that returns as its value the contents of a real valued field on a linked list node. The corresponding function for integer fields is ITMLST.

XTMTBL(IFIELD,NUMBER,ITABLE,IAREA)

XTMTBL is a function that returns as its value the contents of a real valued field on a sequential list node. The corresponding function for integers is ITMLST.

LIST OF FIGURES